UNCLASSIFIED

Defense Technical Information Center
Compilation Part Notice

ADP010755

TITLE: Tools for Optimization and Validation of
System Architecture and Software

DISTRIBUTION: Approved for public release, distribution unlimited

This paper is part of the following report:

TITLE: Development and Operation of UAVs for
Military and Civil Applications [Developpement et
utilisation des avions sans pilote [UAV] pour des
applications civiles et militaires]

To order the complete compilation report, use: ADA390637

The component part is provided here to allow users access to individually authored sections of proceedings, annals, symposia, ect. However, the component should be considered within the context of the overall compilation report and not as a stand-alone technical report.

The following component part numbers comprise the compilation report:
ADP010752 thru ADP010763

UNCLASSIFIED

Tools for Optimization and Validation of System Architecture and Software

C. Fargeon
DGA/DSA/SPMT – Architecture des Systèmes de Drones
2bis rue Lucien Bossoutrot
75015 Paris

Abstract : UAV systems architectures rank among the most complex ones with high safety requirements. Some new software tools have recently emerged that are worth to be known. This lecture was given to advertise four of them on the following topics : 1) Static verification of real time software, to avoid run-time errors, typically what happen for Ariane V ; 2) Simulation of real time architecture to optimize conception and validate the final choice ; 3) edition of command and control software with interpreted properties like on-line automatic reprogramming ; 4) optimization under constraints for continuous and discrete processes.

I - Static verification of real time software

*The software product that is described is developed and maintained by **PolySpace Technologies**, start-up from INRIA, research institute where the research was conducted. The point of contact is Daniel Pilaud, PolySpace T., 655 av. de l'Europe, 38330 Monbonnot-St-Martin, France ; phone / fax : 00-33-476-61-52-60 / 54-09.*

The purpose of the software is to detect run time errors (RTE), compliance with temporal constraints (dead-lock, live-lock), compliance with observable outputs, as well as non-deterministic constructs in real time software. Its technology is based on **INRIA** researches dating back to 1985 on static verification by abstract interpretation. It has been industrialized by PolySpace and experimented convincingly on industrial basis: ARIANE V, Atmospheric Reentry Demonstrator, satellites, railway transport, automotive...

An example of run-time error is the division by zero that can be hidden in a procedure as follows :

```
Procedure foo (
        RR : in Float_64 : F1 : in Float_64 ;
        F2 : in Float_64 : F3 : in out Float_64) is
tmp : Float_64 ;
begin
if (RR /= 0.0) then
        F2 := F1 / RR ;
here, RR may be very small but is not 0.
        ...
        tmp := RR**2 ;
        F3 := F1 / tmp ;
here, tmp may be equal to 0, or F3 may be huge.
end if ;
end foo ;
```

The problem occurs when $tmp < 2^{-511}$ and $F1 > 2^{511}$.

The purpose is to detect and identify the error type, localize it precisely in the source code and depict the error context. In that case, the sofware will issue :
" possible error of correctness condition : denominator must be non zero
computed range : {0 <= [expr] <= 10**10}
in task1.prc_val at "validate.adb" line 24, column 18 : F3 := F1 / tmp ;"

Having in mind that 30 to 40 % of remaining errors are RTE, it is worth while using a tool that automatically detects :

- concurrent access on shared data,
- non initialized variables,
- unreachable code,
- out-of-bound array accesses & buffer overflows,
- arithmetic overflows & underflows (integer, floating point),
- arithmetic exceptions : division by 0, square root (<0),...

The software makes an exhaustive and unambiguous RTE identification through 4 classes of correctness :

- "**certain error**" that are pinpointed by a red color,
- "**potential error**" in orange, that have to be inspected,
- "**always correct**" in green, for which the correctness is proven,
- "**non executable**" in black for unreachable code sections.

An operation causing an error is **necessarily** classified as a potential or certain error.

The principle of the checking is based on polyhedron analysis of always a super-set of the exact solution. The efficiency of the selectivity rate is increased by splitting the state space by several polyhedrons, the use of integer lattices, ...

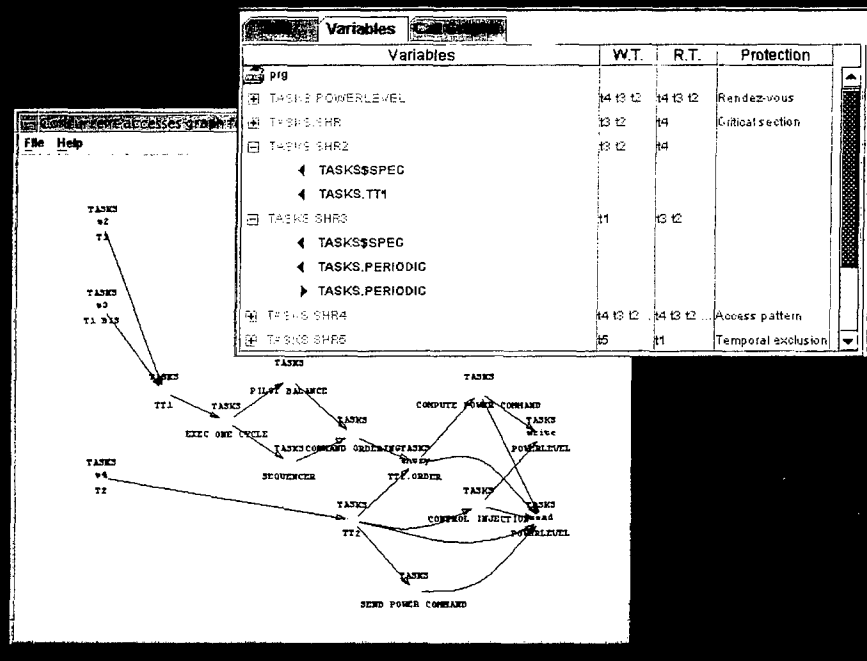
The selectivity rate is important also to save man power. For example, the safety requirements for nuclear power plant induces to have two software specialists to verify the code written by a third one, the ratio is less severe in aeronautical industry but still one might need one verifier for each programmer. If the tool is capable to automatically classify "green" or "red" 85 % of the code,

Paper presented at the RTO AVT Course on "Development and Operation of UAVs for Military and Civil Applications", held in Rhode-Saint-Genèse, Belgium, 13-17 September 1999, and published in RTO EN-9.

PolySpace

PolySpace Viewer

- Exploiting Static Verification results: Global Data



PolySpace

PolySpace Viewer

- Exploiting Static Verification results: source code

```
   if Random.Random then
      VNT(1) := VNT(1) + 1;
   end if;
   if Random.Random then
      VNT(2) := VNT(2) + 1;
   end if;
end PNT;

procedure Foo(Tab: in out Tab T) is
   Max : integer := 0;
begin

   Incr(Counter(4));

   for I in Tab'Range loop

      if Tab(I) > Max then
         Max := Tab(I);
      end if;
```

```
Back
possible failure of correctness condition [non-initialized variable] error range (1<=I<=11)
in "validation 1.adb" line 222 column 12
| if Tab(I) > Max then
|    ^
```

prg Source file: validation 1.adb VALID.SEND_DATA line: 104

that is if the potential errors are reduced to 15 % of the code, then the final inspection needs much less time.

The tool was currently experimented on software of 60 to 100 thousands lines of code. For example, after the crash of Ariane V, it was decided to use this tool to scrutinize the different codes used on the rocket. Ariane V technicians had identified the source of the crash and wanted to prove that this tool was capable to detect the origin of the error in order to apply it to the future generations of codes implied in navigation and guidance pilots. During Ariane experiment, the software tool was thoroughly tested for :

- control and data flow analysis : call graphs, identifying global data, checking initialization,
- interference analysis : finding potential shared memory interference between conflict tasks using the Bernstein criterion,
- scalar range analysis : inferring and checking range conditions for all discrete variables,
- floating-point domain analysis : computing and approximate domain for each floating-point variable, checking the validity of operations.

The second industrial experiment was brought on the Atmospheric Reentry Demonstrator to analyze its critical software (navigation and guidance pilots) including three interacting parallel tasks and 26 thousands lines of code.

To summarize the highlights of this recent tool, the following characteristics can be listed :

- classifies with a very high selectivity rate (between 80 and 95 %),
- works on the following program conditioning : ADA 83 or C ANSI, encapsulation of non-standard constructs,
- the user can tune the duration of the processing from 2 hours, or a day to a week-end,
- the "red tape" consisting of listing all the potential errors is automatically done by the tool.

The time devoted to verification is divided by 5 or 6 and no error or potential one is missed.

II - Optimization of real-time architecture and validation

The second tool presented here has been designed and developed by Michel BARAT from ONERA – research center specialized in Aeronautics and Space Technologies, 27, avenue de la division Leclerc, 92 322 Châtillon cedex, France ; phone / fax : 00 33 146 73 43 88 / 41 41.

The software is a simulator called "SAHARA" that stands for Heterogeneous Architecture Simulator for Agents and Active Resources. SAHARA is a simulator of discrete event process that was originally created for the architecture design of the "pilot associate", Dassault Aviation Program.

SAHARA, initially meant for design, is now moving towards new applications, involving validation and potentially airworthiness certification.

If one examines the tools existing in architecture design, one will come to the conclusion that there is nothing to create and optimize a multiple agent architecture. The only field where solutions are emerging is the area of image processing computers. Solutions are possible in this domain because the structure of the computing is very homogeneous. When agents are heterogeneous, the only way to optimize the design of an architecture is to simulate the process allowing to achieve a solution through an iterative procedure.

One of the main advantages of SAHARA is to be capable of simulating a system during design when components are not well defined or incomplete. The simulator answers this issue by offering a three layer process description : functional, structural, and component level. The three levels of description are the following :

- the functional level can be used to analyze control command of systems with no time delay that is when the available time between two events is always long enough to fulfill the requirements. This stage brings validation of the functional description according to specifications of the system.
- The structural level gives a result if you consider infinite resources. At this stage, you can optimize the options of organization according to goals and constraints.
- The third level describes the precise logical-material system and allows to take into account the impact of time on system behavior. For example, the availability delay of data on data links.

The simulator tests the system according to environment stimulus that can be described with statistical options.

The SAHARA simulator is based on an interpreted Petri net modeling that can take into account asynchronous event, predictable or unpredictable and stochastic events. The elementary level of SAHARA can also be seen as a production rule : If (Condition) Then (action).

The data are described according to their interaction and the way they are exchanged. For instance, a sensor brings out an image but what is of interest on the simulation point of view is not the pixels but

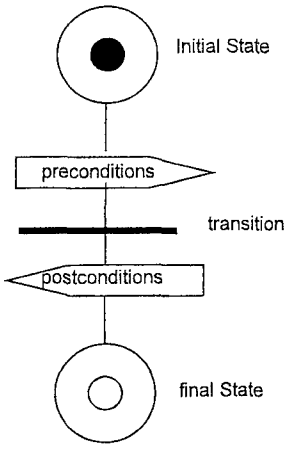
◊ the image size for its impact on communication delay,
◊ and the image pitch for its impact on computation effort.

The system is described using "black boxes" defined by their :

- inputs,
- outputs,
- context,

SAHARA Model

Rule : If condition Then action or
Interpreted Petri Net



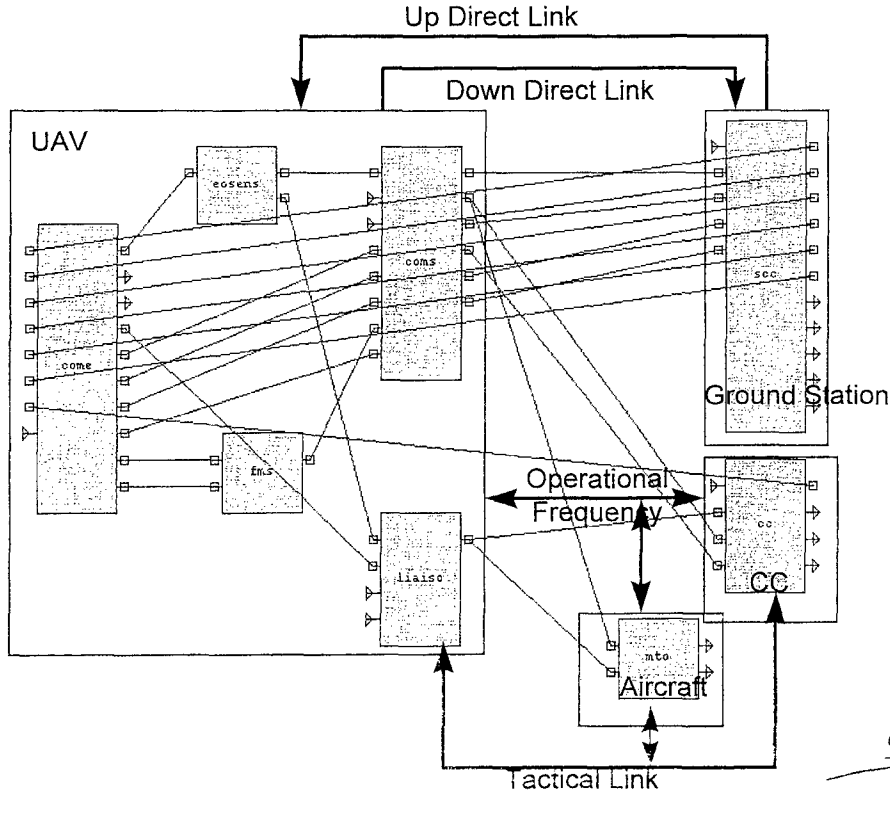
event = data + message

Model : asynchronous
predictable
unpredictable
stochastic

equivalent models
Automatons
Petri Net

ONERA

Structural Level to Logico-material Level



ONERA

- and transformation. The transformation is the propagation of data through the function according to inputs and context.

An example of behavior definition is given below. A behavior can be either deterministic or random.

```
(def_function threadreact
(inputs thread)
(outputs reaction)
(body
 (trigger (thread (equal $thread.level 'urgency))
                (100      immediateReact      (thread)
        ((reaction ((type reflexe))))) )
 (trigger (thread (equal $thread.level 'basic))
                (50      destructionThread      (thread)
((reaction ((type destruction)))))
                (50      avoidanceThread      (thread)
((reaction ((type avoidance)))))) ))
```

What is specified here is :
- for a threat of urgent level : the function applies a determinist behavior and propagates the reaction,
- for a threat of basic level : the function applies a stochastic behavior :
    - in 50% of the cases, the behavior of destroying the threat,
    - in 50% of the cases, the behavior of avoiding it.

One can also depict a periodic behavior like the one that follows :

```
(def_function EOsensor
   (inputs EOcdt)
   (outputs EOframe TacticalFrame)
   (body
      (trigger (EOcdt (frequency (periode 2))
                              (begin (EOcdt (equal
$EOcdt.state 'on)))
                              (end (EOcdt (equal
$EOcdt.state 'off))) ))
                      (100   EOFrameProduct   ()
(EOframe ((length 980 ))))) ) ))
```

In this example, the Electro-Optical sensor produces every 2 units of time an image waiting 980 kbytes.

Delay and synchronized signals can also be taken into account as follows :

```
(def_function evaluationReact
   (inputs thread reaction)
   (outputs reactionNotification defaultNotification)
      (body
            (trigger reaction (maxDelay 10)
                  (begin   (thread   (equal   $
thread.state 'on)))
                  (end   (thread   (equal   $
thread.state 'off)))
                              (outOfDelay   (100
abnormalReact () ((defaultNotification)))) ))
                        (100   normalReact   ()
((reactionNotification)))) ))
```

In this example for asynchronous signal, it is specified that the 2 signals – threat and reaction must be available within a time laps of 10 units. If the time constraint is satisfied, the "notification" of reaction is propagated. If not, the "default" of reaction is propagated.

The user chooses the unit of time according to his need in terms of central processing units, data links, resources. Among resources the user can depict a human operator.

The different applications of SAHARA have given birth to special functions simplifying the user's life :
- interruptible,
- any time function (interruptible, contractual, consultable, latency delay, quality level, context evolution).
- command supervision action.

At function level, one specifies :
- the length of computation through an instruction code measurement,
- the size of the dynamic memory necessary,
- for data exchange, the receiver of the data and the size of the information produced. This size is used to compute the propagation delay via the data link.

At structural level, one looks for the best system composition. For instance, so far as the memories are concerned, they can be defined as :
- local,
- shared,
- reactived,
- buffer (FIFO),...

At this level the simulator can compute the behavior of the system with theoretical resources of infinite capacity.

At hardware level, the logical architecture becomes real with actual resources : CPUs, memories, data links, human agents. The communication channel can be managed according to different strategies :
- either point to point
    - first arrived first served,
    - prioritized ,
    - ...
- or broadcast.

SAHARA has a graphic viewer that gives life to the boxes and links during the simulation for a better understanding of what's going on.

The simulator allows to evaluate :
- for each CPU resources :
    - its instantaneous workload,
    - its maximum load,

- its average load,

- for each data link :
  - the maximum delay of data propagation,
  - its average propagation delay,

- for each memory :
  - its potential conflict,
  - its potential overflow,
  - ...

With all this information, the designer can reshape the architecture of his system and through an iterative process can optimize it.

Today's simulator is based on :
- Work Station Sparc SUN 4 5.6,
- W/WINDOWS,
- Le_lisp and Aida (Ilog)
- MERING 2, Actor and Object library.

The simulator served the following programs :
- in 1989 : to model and simulate the electronic copilot of Dassault Aviation,
- in 1992 : to model and evaluate the UGV "DARDS" architecture for Dassault Electronique, the architecture of this UGV was so safe that never a failure occurred during 7 years of experimentation and demos,
- in 1995 : to accomplish a feasibility study in the context of underwater warfare,
- Nowadays, it is currently used for MAE (Medium Attitude and Long Endurance) UAV architecture design and validation, with the contribution of Sagem and Aérospatiale Matra.

The nest step is to specify improvements of SAHARA in order to enable it to become a useful tool in the process of airworthiness certification of UAVs.

III - Edition of command and control software

The software presented now is called PROCOSA, acronym for the French title "Programmation et Conduite de Systèmes Autonomes" witch can be translated into "Programming and execution monitoring of autonomous systems". This software tool was made-up by Claude BAROUIL from ONERA Toulouse, 2 avenue Edouard Belin, B.P. 4025, 31055 Toulouse Cedex 4, France; phone / fax 00 33 5 62 25 25 61 / 64.

This part is laid out from a paper issued by the author of PROCOSA called "Advanced Real time Mission Management for an AUV" edited in September 1999 for the NATO RESTRICTED symposium on advanced mission management and system integration technologies for improved tactical operations.

The idea behind this tool is to allow automatic reprogramming of autonomous uninhabited vehicles. The code that is automatically issued fulfills the safety requirements and is validated.

An autonomous system is characterized by a high number of functions which are difficult to modify through hardware. All the components at low level are complex and costly to develop. PROCOSA allows to define vehicle behaviors as the organization of control flow and data flow between functions and defines a mission in terms of cooperation between behaviors. Once these functions are set, it is easy to construct a decision level layer.

PROCOSA proceeds into programming a mission by a hierarchical and procedural method : a plan is a partially ordered set of macro actions. A macro action is a particular behavior which can be considered as a sub-mission and is described as such. So, the first step consists in developing those general purpose macro-actions : perception, localization, motion control, payloads...

PROCOSA edits mission plan changes without recompilation : a mission plan refers to behaviors which themselves refer to behaviors and to functions. Changes in mission plans concern only the way behaviors are requested. Functions are never modified during a mission execution. When no function has to be modified, changing a behavior specification is feasible by the mission UAV operator – and not by a system specialist.

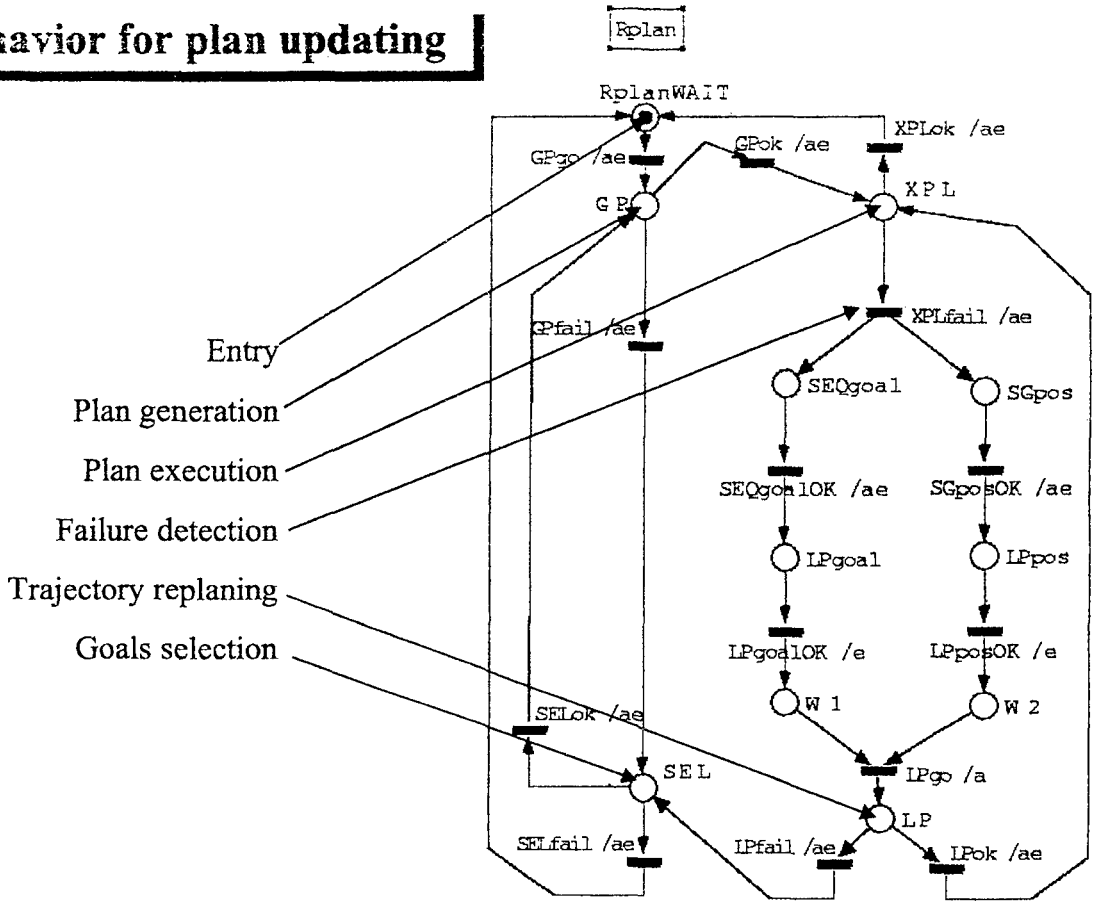
The hierarchical representation simplifies the mission plan design as well as its graphic representation. PROCOSA offers a graphic-based mission plan language, so that a human operator can easily express mission specifications. During mission execution, and when there exists a communication link from the UAV to the operator interface, the same language allows the current mission state to be displayed on the graphic mission representation.

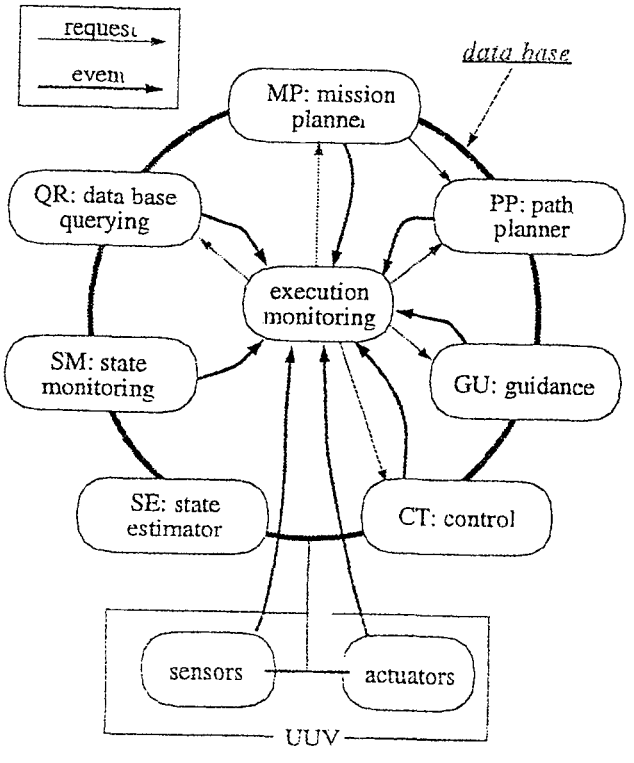
PROCOSA enables easy implementation of failure recovery procedures. This is a very important issue for autonomous UAVs as they have to cope not only with ordinary subsystems failures, but also with unexpected conditions in the environment, which make the current mission plan inadequate. Failure recovery procedures are specific behaviors planned in advance with which behavior or function to activate when such event occurs in such context.

Note that a function may consist in running some plan generation algorithm implemented in some compiled code on the on board computer. The software is based on an extended Petri net formalism (there is no action in the transitions). The Petri player interprets the events according to nets that are read at initialization.

In practice, data manipulated by the player have little chance to have compatible formats. For example, the

Behavior for plan updating





System Architecture

data structure will be different for a picture, a vehicle location, or an image file name. The Petri player needs to tackle this problem. Ideally, the LISP language is necessary that is why an interpreted lisp version has been developed, called TINY.

TINY is specified for on board applications :
- it keeps a safe behavior in case of run-time error,
- it has an easy interface with C,
- it reads several entries (sockets) in parallel.

Events are calling lisp function from TINY package.

The Petri net is in charge of the replanning behavior which is rather simple, even for a real implementation, because the representation sticks only to the replan logic and not to the complete algorithm.

PROCOSA has been recently interfaced with SAHARA in such a manner that it is possible to conduct the architecture and the software design in a unique formalism with validation properties.

PROCOSA has been experimented on various real implementations :
- real time (re) plan itinerary and trajectory of an underwater uninhabited vehicle called REDERMOR to reach areas of interest along a coast ;
- real time deck landing monitoring of a rotary aircraft. In this last case, the implementation was performed on a simulator.

When mission experts have reliably defined behaviors, PROCOSA is on the shelf for efficient execution monitoring implementation.

IV - optimization under constraints for continuous and discrete processes

The fourth topic concerns optimization tools ranking from short term scheduling, vehicle routing and resources dispatching to long term strategic planning. The software is developed and maintained by ILOG, a previous startup from INRIA also. ILOG, no longer a startup, is a well-known company involved in optimal aircraft routing for instance and is located at Bâtiment Orsud, 3-5 avenue Galliéni, 94 257 Gentilly Cedex, France ; phone / fax 00 33 149 08 36 00 / 10.

Since the company has recently edited a white paper on its software suite (see annex), this paragraph will only serve as a short introduction to the white paper.

UAV manufacturers and users will have a number of ways to take advantage of these new optimization algorithms :
- to tackle the problem of monitoring the UAV space allocation especially when UAVs are flying among manned aircrafts,
- to design a UAV, especially the payload allocation since one has to cope with small space for payloads, limited amount of energy, and more globally to optimize the severe cost efficiency objective,
- ...

The key to optimization is problem formulation. ILOG describes problems with decision variables, an objective function and some constraints functions :
- ◊ Decision variables represent parameters that need to be settled. They can be real variables, integer, logical, choices among a set of possible values, etc...
- ◊ The objective function describes the goal. They are of two different types : linear (leading to linear programming), or non-linear. This last category covers a very wide range of tradeoffs.
- ◊ Constraints functions describe logical or physical conditions that the decision variables must obey. In constraint programming, there are not only linear or non-linear expressions, but there can also be rule-based ones like "if A then not B".

The current linear and mixed integer programming algorithms are widely known but suffer limitations (heavy modeling, difficulties to solve large scale scheduling problems, limited means to guide the solution search).

ILOG suite based on constraints optimization combines the power of operation research algorithms with the flexibility and modeling capabilities of expert systems. Constraints-based system scale well into large problems spaces and yield results much faster than other techniques.

The ILOG suite has been used in the domain of transportation, telecommunication, manufacturing, finance, defense, and energy.

Conclusion

UAV systems are complex but will get more and more so, for example one can take the case of a HALE - High Altitude Long Endurance - UAV development. Sophisticated UAVs need to be designed and developed with the most recent software tools to master their complexity.

To have UAV systems expand to new market applications, especially civilian, UAVs will have to undergo severe airworthiness certification procedures. The use of those new tools should reduce drastically the cost of these procedures.

Usually, the cost of these tools is low, what is more demanding is to get into the habit to use them.

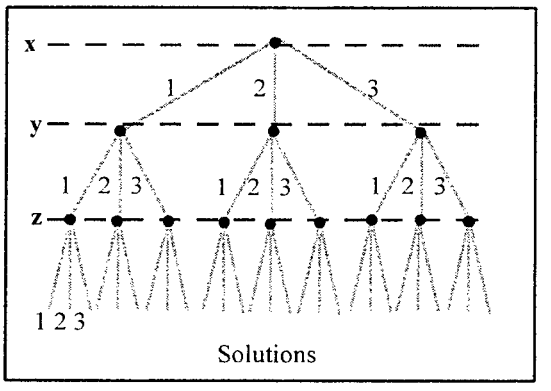
6. How The ILOG Optimization Suite works

6.1. Motivations Behind the Provided Optimization Techniques

This section gives an informal presentation of the main motivations behind the techniques implemented within the ILOG Optimization Suite.

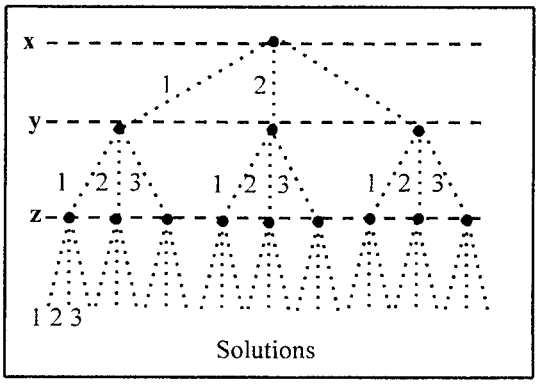
The basic algorithms used in the ILOG Optimization Suite rely on two simple ideas. The first is to explicitly represent the set of values that a decision variable can take. The second is to represent the search for a solution as a tree traversal.

The unknowns of a problem are represented as decision variables. Each decision variable in the ILOG Optimization Suite has a domain or a set of possible values. This domain can be infinite or finite, discrete or continuous. Solving a problem consists of finding the "right" values of its decision variables so that the constraints are satisfied and the objective function is minimized. To do this, the space representing all possible assignments of the variables must be explored. It is convenient to represent this search process in a tree diagram. The nodes represent variables and the branches represent the possible values of these variables.



The search space as a tree

When ILOG Solver follows a branch of a node, it assigns the value of the branch to that variable. Solving the problem consists of finding a path from the root node down to the leaves so that the constraints are satisfied and the objective function minimized.



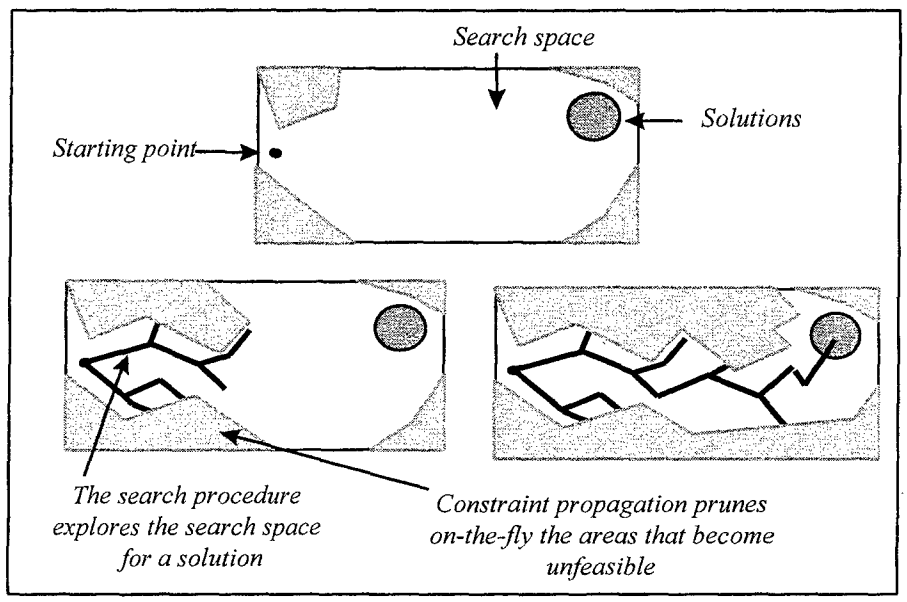
Exploring the search space

As you do not know in advance which branches will lead to a solution, you may need to explore all the possible paths from the root node down to the leaves. However, you cannot blindly try all the values in the domain of the variables, as this is totally unfeasible with real-

world problems. A problem with 10 variables and a thousand possible values for each of them has a search space with $1{,}000^{10}$ possibilities. Years of computational time are required to blindly explore this search space.

Finding a solution to a given problem is often difficult because of the problem constraints. Only a few paths actually satisfy the problem constraints, thereby leading to a solution. One way to overcome this difficulty is to use the problem constraints on the fly, in order to compute the consequences of each choice made on the remaining decisions. This reduces the combinatorial explosion of the search process. The problem constraints are, therefore, used to discover as soon as possible whether the followed path is wrong. This dynamically reduces the search effort still to be carried out, and obtains estimations on how far the search process is from a solution.

The ILOG Optimization Suite provides specific search algorithms that implement these features. Each time a value is tried, ILOG Optimization Suite algorithms deduce the consequences of this modification on the remaining variables, remove from their domains the alternatives that become unfeasible, and thus reduce the computational effort needed to find a solution. This process can be presented with the following diagram.



Search process

6.1.1. Domain reduction

As ILOG Solver, the core engine of the ILOG Optimization Suite, searches for a solution, it removes from the variable domains the values that are no longer feasible. Eliminating values that are no longer part of a solution is called *domain reduction.*

When the domain of a decision variable is modified, ILOG Solver uses the constraints to compute the consequences of this decision and remove from the domains of other variables the values that cannot satisfy the constraints and, therefore, cannot be part of a solution. The process of computing the consequences of the modifications and reducing the domains of the variables is called *constraint propagation.*

By reducing the domains on the fly, the ILOG Optimization Suite rapidly reduces the "combinatorial explosion" problem and avoids significant computational loads. The domain reduction process is implemented using the following principles. When you reduce the domain of a variable, you refine the information known about this variable. The ILOG Optimization Suite uses this information to update the set of possible alternatives for the remaining variables. This can be illustrated with the scheduling of maintenance activities. If you reduce the possible

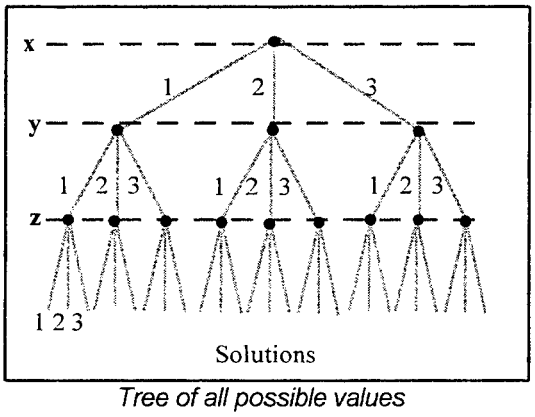
starting time of an activity so that it must be performed between June 3 and 5, you can deduce that all preceding activities must be finished by June 3 at the latest, and that all following activities start on June 5 on the earliest.

6.1.2. An example

Consider the following simple problem:

```
Find integer values for x, y and z such that:
        x ∈ [1,3], y ∈ [1,3], z ∈ [1,3]
            y<z, x-y=1, and x≠z.
```

The combination of all the assignments of x, y, and z is represented as a tree whose nodes represent variables and whose branches represent the possible values of those variables. Nodes at the same level represent the same variable.



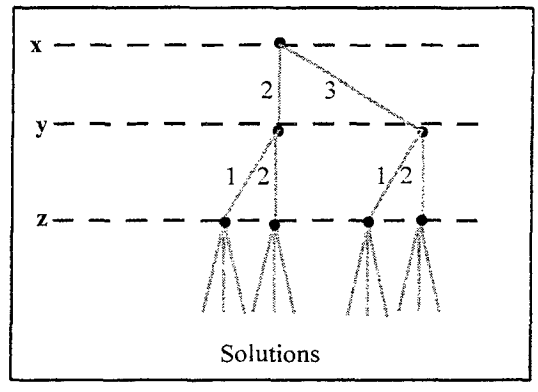
Tree of all possible values

This tree represents the search space - the space of all possible assignments. The set of paths from the root node to the leaves represents the combination of all possible assignments to x, y, and z. A possible path does not necessarily satisfy the problem's constraints, and therefore may not be a solution.

At this point, you can say that if there is a path from a root node down to the leaves that satisfies the constraints, then it is a solution to your problem. You can now apply the principles introduced in the previous sections.

6.1.2.1. Initial propagation

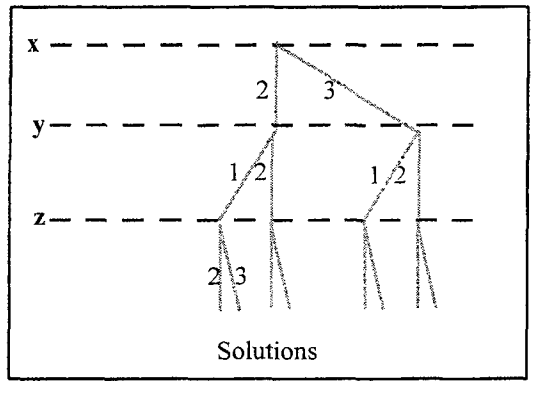
Consider first the constraint x-y=*1*. This constraint implies that the smallest value of x must be one unit higher than the smallest value of y. Therefore, *1* cannot be a possible value for x and is removed from its domain. The domain of x is now *[2, 3]*. Similarly, the maximal value of y must be one unit smaller than the maximal value of x, and the domain of "y "becomes *[1, 2]*. The domain of z remains unchanged since no constraints involve z at this point. You can update the tree by pruning the branches corresponding to the removed values. You obtain the following tree:



Pruned tree

If you now add the second constraint, $y<z$, the minimal possible value for z is 2, because z must be at least one unit higher than the minimal possible value of y. The domain of z is then reduced to *[2, 3]*, while the domain of y stays unchanged.

The updated tree is as follows:



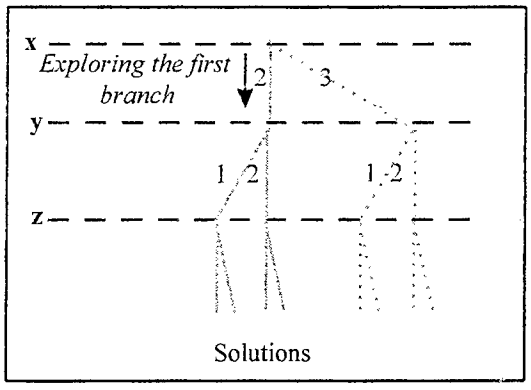
Pruned tree

When the third constraint $x \neq z$ is added, we can't apply any reduction on the domains of x and z. For now the domains of these variables remain unchanged.

The initial propagation of the constraints reduces the variable domains and prunes the search tree. However, the variables x, y, and z still have several possible values in their domains. You use a search procedure to explore the remaining branches in looking for a solution.

6.1.2.2. Finding a first solution

The tree search procedure will try the different values from the domain of these three variables. Assuming that the search procedure selects the variable x to start with and follows the first branch, when ILOG Solver follows the first branch, namely branch 2, it assigns the corresponding value to the variable. Here, the value 2 is assigned to x.



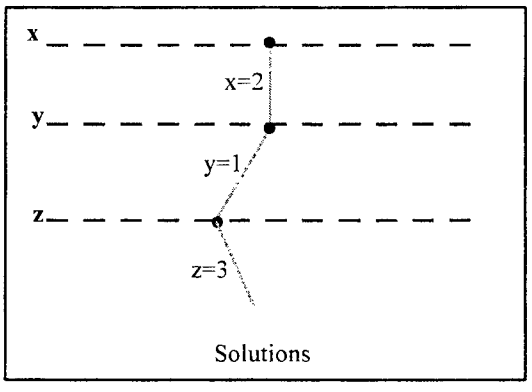
Exploring the search tree

At this point, constraints are automatically used by ILOG Solver in order to further reduce the other variable domains, if need be. As the domain of x has changed, the constraints x-y=1 and $x \neq z$ are activated:

- The constraint x-y=1 reduces the domain of y to the value 1. 1 is therefore assigned to y, as it is the only remaining possible value.

- The constraint $x \neq z$ removes 2 from the domain of z. The variable z now has one possible remaining value, namely 3. The value 3 is therefore assigned to z.

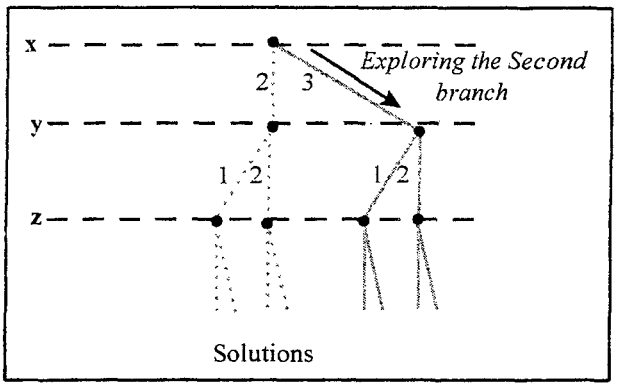
The 3 variables x, y, and z are all assigned and the constraints are satisfied. Therefore, the first solution to the problem is x=2, y=1, and z=3. ILOG Solver finds this solution after making only one choice.



The solution

6.1.2.3. Looking for other solutions

If you are looking for another solution, ILOG Solver backtracks. This means that it undoes the last decision it made and explores another branch of the tree. In the example, that decision was x=2. When ILOG Solver backtracks, the decision is undone, together with all its consequences. The domains of all the variables are restored to the state they were in before the decision x=2 was made. The variable domains are now $x \in [2,3]$, $y \in [1, 2]$, and $z \in [2, 3]$, and you are once again at the node labeled by x.



Backtracking and exploring the second branch

A new branch is followed, corresponding to the decision $x=3$, and the constraints are propagated once more:

- The constraint $x-y=1$ deduces that y must be equal to 2.
- The constraint $x \neq z$ removes 3 from the domain of z leaving 2 as the last possible value. 2 is therefore assigned to z.
- As the variables y and z have been modified, ILOG Solver activates the other constraints involving those variables. In this case, it is the constraint $y<z$. As the value of z is 2, this constraint sets the maximum of y to 1. Because y has yet to equal 2, an inconsistency is raised and triggers a backtrack. ILOG Solver deduces that there is no possible solution down this branch.

6.1.3. Efficiency of constraint propagation

The domain reduction process is used to reduce the search effort required to find solutions. However, the efficiency of this process, based on simple principles, depends heavily on its computing power and its capabilities to evaluate the global impact of decisions and to remove unfeasible alternatives as soon as possible from the search space. For this, the ILOG Optimization Suite integrates strong optimization algorithms. These algorithms can solve constraints, compute the global consequences of decisions, and drive the search process toward the targeted solutions.

6.2. The Provided Types of Variables and Constraints

ILOG Solver provides a rich set of variable types, constraint classes, search strategies and optimization algorithms. Objects are easily used directly by developers to model and solve resource allocation problems. The list of classes provided includes (and is not limited to) integer variables, floating point variables, Boolean variables, set variables, $=$, $\leq$, $\geq$, $<$, $>$, $+$, $-$, $*$, $/$, scalar product, subset, superset, union, intersection, member, Boolean or, Boolean and, Boolean not, Boolean If-Then, cardinality, distribute, extensively defined relations, trigonometric functions, power, square root, logarithm, exponential, as well as meta-constraints (conjunction and disjunction of constraints, order among constraints). You can also add new types of constraint to ILOG Solver by deriving new C++ classes.

To help you explore the search space, ILOG Solver provides a branch-and-bound algorithm and a backtracking search –chronological and non-chronological– together with a large number of search strategies. ILOG Solver also provides C++ classes and functions that implement non-deterministic programming. The latest functions can be used to implement any specific tree search algorithm.

6.3. Searching

Domain reduction is not enough to deduce the values of all the variables of a problem. For example, the problem under consideration may have several feasible solutions. A complementary search procedure is used to find an explicit solution to the problem in question.

6.3.1. Available procedures

With the ILOG Optimization Suite, you can explore the search space in several ways:

- You can use Solver to explore the tree search exhaustively, find the best solution to the problem, and prove its optimality.

- If you are looking only for a feasible solution or a set of alternative solutions to your problem, you can also use ILOG Solver to compute such solutions.

- You can also use ILOG Solver to gradually improve or repair a solution. The improvement process can be stopped at any time, and the best result found so far can be returned.

6.3.2. Searching and constraint propagation

The ILOG Optimization Suite intensively uses the constraints during a tree search to compute and propagate the consequences of each decision made. The constraint propagation engine of the ILOG Optimization Suite carries all constraints together during the search process. It computes the consequences of each decision taken by the search procedure or the user, tries to find inconsistencies as soon as possible, reduces the number of alternatives to be considered during the search, and drastically reduces the computational effort needed to find a solution.

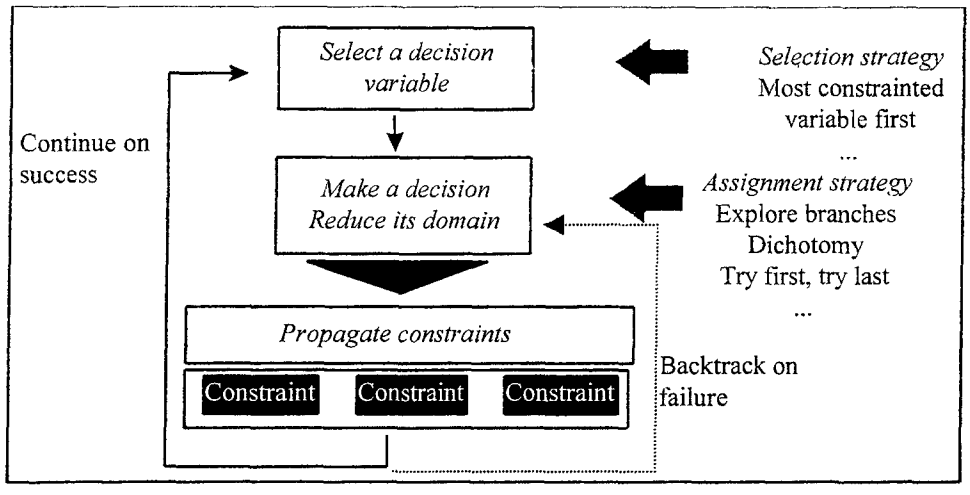
The exploration of a branch may lead to an inconsistency, that is, to a state where some variables have an empty domain. In this case, no solution is possible, and ILOG Solver automatically backtracks in order to follow another branch. Since no one knows which branch of the tree leads to a solution, ILOG Solver explores all the branches until one of them leads to a solution. In the example's case, the second branch of the tree does not lead to a solution. The first solution is the unique solution to the problem.

At each new node of the search tree, the constraints are used to reduce domains of decision variables. This domain reduction has two important consequences:

- The search strategy is improved because the set of alternatives coherent with the current partial solutions is dynamically and automatically maintained. Thus, alternatives that are obviously impossible are not considered.

- Inconsistencies are discovered early in the search process. As soon as a variable domain is empty, ILOG Solver knows that no solution can be obtained from the current branch.

These consequences are powerful enough to prune very large parts of the search tree, leading to superb performance. The search algorithm described here is implemented using the ILOG Solver function `IlcGenerate`. The order in which the variables are considered can be dynamically computed. For instance, a user can choose, as the next variable, the one that has the least number of possible values in its domain. This ordering among the variables does not affect the solution's validity but it can be very important for improving performance. The following figure illustrates the architecture of the search procedure.



Architecture of the search procedure

6.3.3. Implementing new search procedures

ILOG Solver also offers programming functions to create and explore the search tree in new ways. The tree-search algorithm described above can be implemented by two functions: branch and generate. Branch tries the different possible values from the domain of a variable. Generate takes an array of variables as arguments and calls branch to try values from the domains of those variables according to the following process:

1. As long as there are any unknown variables

2. Choose one of these variables

3. Choose a value to assign to this variable

4. Propagate the effect of that assignment; go back to step 1

There is a hidden problem in this description. In general, you do not know which value in the domain of a variable leads to a solution. Assigning a value to a constraint variable must be seen as a guess. If after this assignment an inconsistency is detected, it must be undone, and another value must be tried. This backtracking improves the previous algorithm as follows:

1. As long as there are any unknown variables

2. Choose one of these variables

3. Choose a value to assign to this variable

4. Assign the chosen value to this variable and memorize in parallel so that if a contradiction is detected afterwards, remove the tried and failed value from the variable domain and try another value; go back to step 1.

To express these search algorithms based on this try-backtrack on failure-retry mechanism, ILOG Solver provides goal programming.

The goals **Branch** and **Generate** can be implemented in the following pseudo-code:

```
ILCGOAL1(Branch, IlcIntVar, x){
 if (x.isBound()) return 0;
 IlcInt a = StrategyChooseValue(x);
```

```
  return IlcOr(x == a, IlcAnd(x != a,
                              Branch(x)));
}
ILCGOAL1(Generate, IlcIntVarArray, vars){
  int i = StrategyChooseVar(vars);
  if (i == -1) return 0;
  return IlcAnd( Branch(vars[i]),
                 Generate(vars));
}
```

The important property of these goals is that they can be used to implement search algorithms other than the one used in `IlcGenerate`. In other words, they give the ILOG Solver user greater control over the actual search for a solution. Thanks to this extensibility, customers can incorporate domain-specific knowledge about the interaction among goals and constraints to provide very effective search accelerators. In fact, any tree-search algorithm can be expressed in ILOG Solver using goals.

6.4. About Constraint Propagation Algorithms

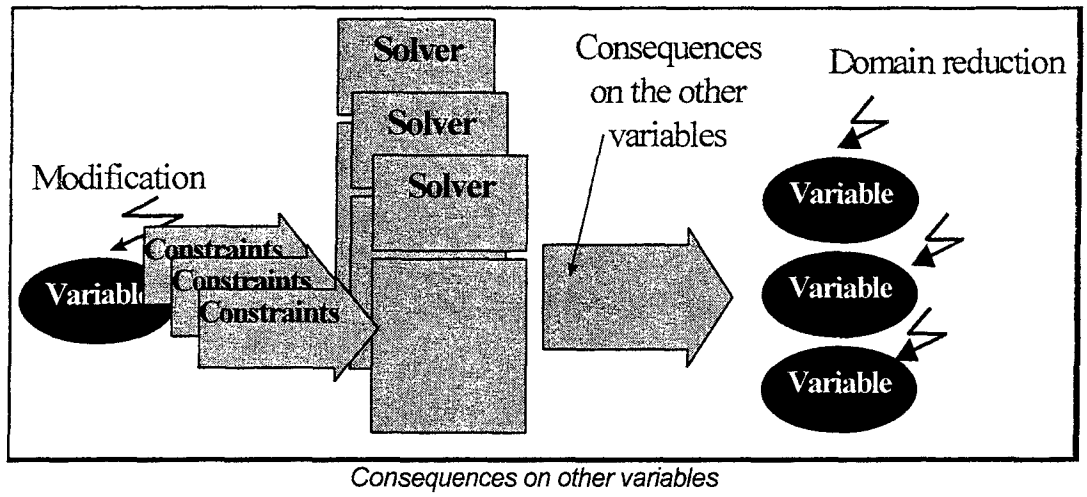
The constraint propagation algorithms carry out the reduction of the variable domains. The following sections describe how algorithms are implemented and perform this domain reduction.

Each class of constraints has its own algorithm, called the *solving algorithm*. An object, pointing to the decision variables that are involved in a constraint, implements the solver algorithm for this constraint. This algorithm removes from the domains of decision variables those values that cannot satisfy the constraint and therefore cannot really participate in a solution. Thus, each constraint keeps the variable domains consistent with its relation. Each constraint maintains its arc consistency. The word arc-consistency is used to emphasize the fact that a constraint is seen as an arc that links the variables together.

The solving algorithm of a constraint can also look ahead to the remaining possible values of the decision variables and check whether the constraint can be verified with these values. For example, take the constraint $x = y$ where x is smaller than *5* and y is greater then *10*. The solving algorithm immediately deduces that x and y will never be equal, because there is no intersection between the possible values of x and y. The ILOG Optimization Suite immediately reports this inconsistency.

6.4.1. Activation of the Solving Algorithms

When a constraint is posted on variables, ILOG Solver stores this constraint and activates its solving algorithm to update the domains of the variables in question. ILOG Solver also activates the solving algorithm of this constraint each time the domain of a relevant variable is modified. The solving algorithm then computes the consequences of this modification on the other variables involved by the constraint. This is the basis of the constraint propagation process described below.



Consequences on other variables

6.4.2. Constraint propagation

When the domain of a variable is modified, ILOG Solver activates the algorithm associated with the constraints that involves this variable. To stay arc-consistent, a constraint algorithm may reduce the domains of some other variables. This variable may be involved in other constraints that are then also activated for consistency. This activity is known as constraint propagation. The constraint propagation process is repeated until all the constraints are arc-consistent and no more information can be deduced from the modification. The constraint propagation process makes the constraints collaborate together in order to deduce as much information as possible from a modification of the domain of a variable. It is also one of the most important activities that the ILOG Optimization Suite handles automatically for you.

6.4.3. Arc-consistency algorithms

Generally in constraint-based optimization, various techniques are employed to remove values. ILOG Solver uses an extension of the AC-5 arc-consistency algorithm proposed in [DH]. The original AC-5 algorithm only handles integer decision variables, whereas ILOG Solver also handles set, Boolean and floating-point variables. This algorithm has been implemented very carefully because it is at the core of ILOG Solver's performance and accuracy. For instance, the constraints are not necessarily considered whenever the variable domain is modified. They are propagated only under certain conditions. For example, the constraint $y < z$ is activated only when the maximum possible value for z or the minimal possible value for y changes. No other reductions of the two domains would lead to additional reduction of any of the domains. For this reason, the constraint does not need to be activated. When compared to other constraint propagation implementations, ILOG Solver is seen to be more than 2,000 times faster [P.L].

6.5. Using Simplex Algorithms with ILOG Planner

6.5.1. Why simplex algorithms?

The family of simplex algorithms has proven to be very efficient in real-life applications that let you build a linear model of your problem; that is, when the model contains mainly linear constraints. ILOG Planner has been designed to profit from the efficiency of these algorithms in applications in which a lot of constraints can be placed in a linear form. ILOG Planner can significantly speed-up the ILOG Optimization Suite. This allows for addressing more efficiently particularly big or difficult applications.

ILOG Planner includes primal, dual and network simplex algorithms, and benefits from CPLEX algorithmic know-how.

6.5.2. Basic principles of ILOG Planner

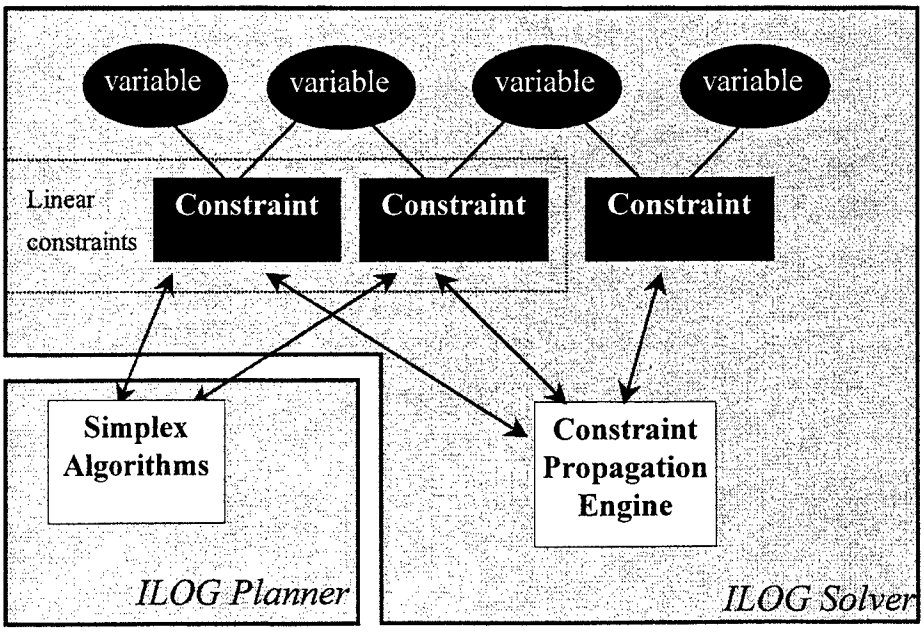
ILOG Planner shares the variable and constraint objects of ILOG Solver, allowing it to benefit from the elegant C++ syntax provided by Solver.

The way ILOG Planner and ILOG Solver work together is based on the traditional branch and bound procedures used to solve integer problems in linear programming. Basically, a linear relaxed problem is used to provide an approximation of the optimal solution. For a minimization (maximization) problem, this approximation provides a lower (upper) bound of the cost function. Then, the optimal relaxed solution computed by the simplex algorithm may be used to guide the search procedure.

6.5.3. How ILOG Planner works

ILOG Planner implements the following mechanisms. First, you notify ILOG Planner of the linear constraints in the problem to be solved. Then, the simplex algorithm of ILOG Planner computes the optimal solution (if any) of the problem, and sends the optimal cost value to ILOG Solver, which updates the domain of the cost variable accordingly. Finally, a search procedure built with the search facilities of ILOG Solver may use the optimal relaxed solution provided by ILOG Planner as an approximation of the optimal solution. Late during the search, when the search procedure deduces or tries new bounds on a variable, ILOG Solver sends the new bounds to ILOG Planner, which updates the relaxed solution according to this bound modification and computes the new minimum value of this cost function.

You don't need to model your problem using only linear constraints to take advantage of ILOG Planner. That's why ILOG Planner is particularly well suited to problems for which linear programming techniques are not applicable but involve a lot of linear constraints.



Revised simplex procedure

7. Benefits of The ILOG Optimization Suite

Constraint-based optimization and especially the ILOG Optimization Suite have been successful implemented in very diverse projects. In this section, we analyze the current successes of constraint-based optimization, and characterize the types of problem in which it can be applied.

7.1. Interactive and Reactive Search Capabilities

The ability to define constraints and variables and find a solution is not enough for a lot of practical applications. Users may need to intensively interact with the planning applications. They would need, for example, to generate several plans, explore scenarios, run simulations and what-if analyses, and add or relax constraints. In other applications, *reactivity* is a key issue. It means that the planning applications must be able to update plans according to new data, new constraints or new priorities.

To satisfy these requirements, the ILOG Optimization Suite provides a way to *manage* constraint models. ILOG Solver provides a mechanism, the *manager*, which allows adding and removing constraints, completely or partially storing the current solution, and restoring the previously stored partial or complete solution.

Moreover, another benefit of the ILOG Solver manager is the possibility of writing advanced optimization algorithms relying on a limited search in a subtree, iterations on local optimization, problem decomposition, etc.

7.2. Programming Search Strategies

The core technology of the ILOG Optimization Suite has several benefits. The ILOG Optimization Suite actively uses constraints and removes the unfeasible alternatives from the variable domains on the fly. In this way, the ILOG Optimization Suite converges on solutions much faster, systematically reducing the number of possibilities to be explored. The domains of the variables are constantly updated during the search, providing very useful information for refining the search strategy.

A more general advantage of the search technique used in the ILOG Optimization Suite is that the search strategy is independent of the constraints and can be programmed. The next decision variable to be explored is identified dynamically once all the constraints involved in the current node have been propagated. The order in which the nodes of the search tree are explored is therefore dynamic and programmable with the ILOG Optimization Suite.

7.3. Improving Solutions

With many problems, you can find an initial solution very easily. However, to find an optimal solution and verify its optimality, you usually need long running times. For these problems, you can easily implement a search using the ILOG Optimization Suite algorithm that generates a preliminary solution very quickly and then gradually improves the result. This approach can be stopped at any time, with the best result found so far returned.

7.4. Exploiting Problem Knowledge

As a user of the ILOG Optimization Suite, you can make the search itself more efficient by exploiting knowledge about the problem. For example, the order in which the nodes are explored in the tree may be very important. Consider the timetabling application for mainframe operators at the Banque Bruxelles Lambert. Some parts of the year are more difficult to schedule than others. Normally, many engineers want to take their vacations in the summer

and around Christmas. It seems reasonable, therefore, to assign the shifts corresponding to the Christmas week first, as this is the week in which the problem is the most difficult to solve. This kind of information is called "strategic knowledge," since it deals with the way the problem should be solved.

Strategic knowledge is quite easy to use with the ILOG Optimization Suite. In fact, if you look at the basic search algorithm, The ILOG Optimization Suite enables you to control the order in which the variables are selected. This principle can be applied to dramatically improve performance on very complex problems.

7.5. Integration of Operations Research Algorithms

A very attractive property of the ILOG Optimization Suite is that it easily integrates specialized algorithms for solving a given problem. In other words, constraint-based optimization is a framework for cooperating problem-solving algorithms. Algorithms are implemented as constraint solvers, and the custom algorithms and standard solvers communicate via the variables.

There are several algorithms already integrated in the ILOG Optimization Suite. This section now describes two of these algorithms: the revised simplex and the edge-finder.

7.5.1. Revised simplex for linear constraints

Linear Programming algorithms with the simplex procedure are widely used in solving linear problems such as liquid blending and production planning. ILOG Planner complements ILOG Solver by providing a simplex solver to handle problems that involve many linear constraints or can be represented with linear models. ILOG Planner handles linear constraints while ILOG Solver algorithms handle logical constraints.

ILOG Planner solves the problems that occur in the standard linear programming approaches: integer and mixed-linear integer programming. When looking for solutions, ILOG Planner relies on the search procedures of ILOG Solver. It benefits from all the flexibility of the ILOG Solver functions in driving the search, implementing search strategies and solving directly logical constraints.

7.5.2. Resource constraints for scheduling applications

For scheduling problems involving time and sequencing, the edge-finder algorithm is implemented in the finite-capacity resource constraints of ILOG Scheduler. This algorithm is one of the most successful operations research algorithms for rapidly updating time windows of activities submitted to resource constraints. This innovation incorporates the efficiency of this algorithm with the flexibility of constraint-based optimization.

7.5.3. User-defined global constraints

You can define global constraints, that is, constraints shared by a set of variables. An example of this is a resource allocation problem such as assigning cashiers -to work areas (i.e., cash registers) in a department store.

Two cash registers each require two people. In ILOG Solver, the model for this problem includes two decision variables, one for each person. The possible values for these variables are the cash registers. The only constraint in the problem is that no variable pair can be assigned the same cash register. This is simple and intuitive.

In some systems, this restriction might be expressed by stating a different constraint for each variable pair. However, if n is the number of people, must you then have n^2 constraints? Stated

this way, the problem model is unnecessarily complex and consumes an enormous amount of memory and processing time. Indeed, if you have 1,000 people in a chain of stores, you would have to create 1 million constraints.

An alternative provided by ILOG Solver is to use only one global constraint and share it with all the variables. The space required by the constraint is linear with respect to n.

7.6. What About Other Techniques?

It can be argued, though, that the problems tackled by the ILOG Optimization Suite are intractable. (They are often known to be NP-hard.) In other words, certain problems may take an exponential amount of time to solve. Fortunately, this issue rarely occurs in practical real-world situations.

First of all, our experience at ILOG has shown that such "intractable" problems are already partly solved manually. Usually the solutions are not automated but the problems cannot be represented satisfactorily with less-expressive problem solvers. Indeed, these problems require all the expressive power of the ILOG Optimization Suite in order to be represented accurately enough.

Those solutions that can be represented with less-expressive packages are already solved by those packages, and consequently the solution is "good enough," although it may be improved upon. The ILOG Optimization Suite can generate exactly the same solution using the same imprecise problem definitions. Thus, although a purely optimal solution to the problem is not currently needed, using the ILOG Optimization Suite still allows more maintainable code and easier expansion to meet future needs.

Another important issue is the environment in which resource allocation systems must be integrated. Most ILOG Optimization Suite applications are not batch applications that read data and then produce a solution. They are decision-support applications. Indeed, they often present an interactive graphical user interface when displaying a solution. In these applications, users want to see the current solution and monitor the search algorithm. They may even want to stop the search, remove or add constraints, and start the solution search again at a given point.